Using Divide and Conquer For Space Partitioning In Procedural Town Asset Composition

Raysha Erviandika Putra - 13524050[1,2]
Program Studi Teknik Informatika Sekolah
Teknik Elektro dan Informatika
A. Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13524050@std.stei.itb.ac.id, [2]rayshaervi557@gmail.com

***Abstract*— Procedural town generation often becomes messy when asset placement is handled directly from random positions. This paper presents a simpler method that treats a town as a grid of square blocks and applies a Divide and Conquer subdivision process when a road cutter intersects a block. A block that is not touched by a road is kept as a large leaf. A block that is touched by a road is divided into four smaller blocks until it reaches the minimum cell size. The final leaves are then used to spawn roads, ground, parks, houses, medium buildings, or larger buildings. The implementation is made in Unity using a small set of scripts and Kenney Modular Buildings as visual assets. The result expected from this work is not a realistic city simulator, but a clear adaptive town composition prototype where road position changes the spatial subdivision and the subdivision result changes the asset scale.**

Keywords*— Divide and Conquer, procedural generation, town generation, adaptive subdivision, Unity

I. Introduction

The project research talks about the idea of dynamic town generation using a different approach. This starts from a simple layout rule where usually roads occupy part of a town area, and the remaining space should be divided into building lots that match the road shape. A fixed uniform grid can handle this, although it creates the same level of detail everywhere. A fully random placement method also makes it difficult to explain why a large building appears in one location and a small house appears in another.

The generator represents the town on the Unity XZ plane. The town is first divided into square root blocks. Road objects are placed as BoxCollider components. Every root block is checked against those road colliders. A clean block becomes a final leaf. A touched block is split into four smaller squares, then each child is checked again. This continues until the minimum cell size is reached.

The resulting tree records where detail is needed. Large regions remain intact when a road stays outside their boundary. Regions around a road receive deeper subdivision. The final leaves are used directly during asset placement. A leaf with a side length of four grid cells can hold a large building. A two-cell leaf can hold a medium building. A one-cell leaf can hold a small house or a road surface.

This paper examines how can Divide and Conquer can be applied to spatial subdivision so that a road placement controls procedural town asset composition. The scope is limited to the subdivision algorithm, road intersection, asset tier selection, and a small Unity implementation. Urban simulation, traffic flow, terrain height, and zoning rules are outside the current implementation.

II. Theory Foundation

A. Divide and Conquer

Divide and Conquer solves a problem by breaking it into smaller instances of the same general problem, solving those instances, and assembling the result. The method can be described through three steps, that is divide, conquer, and combine [1]. The divide step produces smaller subproblems. The conquer step solves each subproblem directly or recursively. The combine step gathers the subsolutions into the final result.

The three steps can be applied to this generator without changing their meaning. The current square block is the problem instance.

1. Divide, splitting the square into four equal children is the divide step.
2. Conquer, calling the same function for each child is the conquer step.
3. Combine, the leaf list produced by those calls is combined during the spawning pass, where every final region receives ground, road, or building geometry depending on the size of an occupied grid

B. Quadtree Spatial Subdivision

A quadtree is a hierarchical data structure used to divide a two-dimensional region into smaller areas. Each internal node has four children, and each child represents one quadrant of its parent. Finkel and Bentley introduced the quadtree as a structure for organizing two-dimensional spatial data [2].

A quadtree node can be described by its position, size, depth, and children. The root node covers the complete spatial domain or one large starting block. When a node is divided, its area is split evenly into four child regions. These children cover the same area as the parent without overlapping each other.

Subdivision does not need to occur at the same depth across the entire region. Each branch can stop independently based on a condition defined by the application. Areas that

require little spatial detail can remain as large leaf nodes. Areas that contain more complex boundaries can continue into smaller nodes. This property is known as adaptive spatial subdivision.

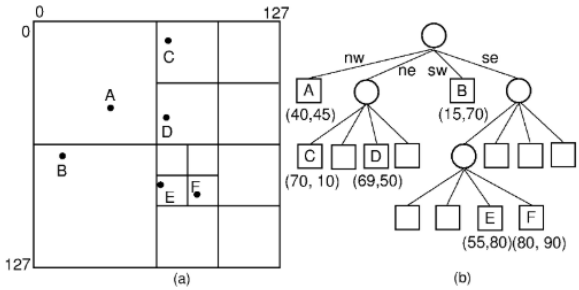

Fig 1. Quadtree Spatial Subdivision
Source: Adapted from OpenDSA, "The PR Quadtree" [2].

In this project, the town layout is represented on the Unity XZ plane. Each starting block contains a square region of grid cells. A road is represented by a BoxCollider. The generator checks whether the bounding area of a node intersects one of the road colliders. A node without an intersection becomes a final leaf. A node that intersects a road is divided into four smaller nodes when its size is still above the minimum cell size.

Each child is tested independently after subdivision. Children outside the road area can stop and become building lots, while children that still intersect the road continue to the next depth. This creates smaller partitions around road boundaries and keeps larger partitions in areas that are unaffected by roads.

The final leaf size is also used during asset placement. A leaf with a side length of four grid cells can receive a large building. A leaf with a side length of two grid cells can receive a medium building. A one-cell leaf can receive a small house or a road surface. The quadtree therefore acts as the spatial structure that connects road intersection with the scale of the generated town assets.

C. XZ Plane Intersection

The building blocks are axis-aligned squares. The road cutters may be rotated. The implementation reads the BoxCollider center, size, world scale, right vector, and forward vector. It then performs a two-dimensional separating-axis test on four axes: the world X axis, the world Z axis, the road right axis, and the road forward axis.

For each axis, the block and road are projected into one-dimensional intervals. The intervals overlap when the distance between their projected centers is less than or equal to the sum of their projected radii. The road intersects the block when all four axes overlap. This method supports straight, perpendicular, and diagonal road colliders without converting the complete scene into a physics simulation.

Unity Bounds and collider bounds provide the basic world-space representation used by the generator [4]. The custom projection test is kept because a rotated BoxCollider cannot be represented precisely by one axis-aligned rectangle on the XZ plane. The road surface is clipped against the final road leaf before rendering, which reduces the rectangular overdraw around diagonal roads.

D. Unity Prefabs and Procedural Asset Composition

A prefab in Unity is a reusable asset that stores a GameObject together with its components. Prefabs are used oftenly in generation because a program can create objects during generation without constructing each GameObject manually.

A generator only needs a reference to the prefab and its instance criteria. In this project, prefabs represent the visual assets placed inside the final quadtree leaves. Building prefabs are grouped into small, medium, and large categories through SimpleTownPalette. After subdivision finishes, the generator reads the size of each leaf and selects a prefab from the matching category. Kenney Modular Buildings is used as the visual source. The pack contains 100 files and is released under Creative Commons CC0 [3].

The field named cells stores the side length of a node in grid cells. A value of four represents a 4 x 4 cell region. A value of two represents a 2 x 2 region. A value of one represents a single cell.

III. IMPLEMENTATION

A. The Project

The implementation uses three scripts inside a Unity 6 project. SimpleOctreeTownGenerator.cs contains the quadtree subdivision and object spawning. SimpleTownPalette.cs stores prefab references. SimpleTownGeneratorEditor.cs adds Generate and Clear buttons together with the generation values. The main parameters define the town grid, the initial block size, the smallest allowed leaf, and the road colliders.

The recursive state is stored in a small Node class. The cells field is the side length of the region in grid cells. The bounds field stores its world-space location and size.

```
class Node
{
    public readonly Bounds bounds;
    public readonly int depth;
    public readonly int cells;
    public bool isRoad;
    public readonly List<Node> children = new();
}
```

B. Root Blocks Creation

Generate() function scans the town in steps of startBlockCells. Each position creates one square root node and sends it to the recursive method. Mathf.Min keeps the final blocks so that they remain inside the configured town size.

```
for (int x = 0; x < townCellsX;
     x += startBlockCells)
{
    for (int z = 0; z < townCellsZ;
         z += startBlockCells)
    {
        int cells = Mathf.Min(
```

```
            startBlockCells,
            Mathf.Min(townCellsX - x,
                      townCellsZ - z));

        Node root = CreateBlockNode(
            x, z, cells, 0);
        BuildNode(root);
    }
}
```

CreateBlockNode() function converts grid coordinates into a world-space bounds value. The block center is calculated from the cell index, cellSize, and block side length. All subdivision decisions after this point operate on bounds rather than separate grid coordinates.

C. *Recursive Subdivision*

BuildNode() contains the full Divide and Conquer decision. A clean node is stored immediately. A touched node at the minimum size becomes a road leaf. Larger touched nodes are divided and each child is processed with the same method.

```
void BuildNode(Node node)
{
    bool touchedByRoad =
        IntersectsAnyRoad(node.bounds);

    if (!touchedByRoad)
    {
        leaves.Add(node);
        return;
    }

    if (node.cells <= minCells ||
        node.cells <= 1 ||
        node.cells % 2 != 0)
    {
        node.isRoad = true;
        leaves.Add(node);
        return;
    }

    SplitIntoFour(node);
    foreach (Node child in node.children)
        BuildNode(child);
}
```

The split creates four children with half of the parent side length. Their centers are placed in the four XZ quadrants.

```
void SplitIntoFour(Node node)
{
    int childCells = node.cells / 2;
    node.children.Add(
        CreateChild(node, -1, -1, childCells));
    node.children.Add(
        CreateChild(node,  1, -1, childCells));
    node.children.Add(
        CreateChild(node, -1,  1, childCells));
    node.children.Add(
        CreateChild(node,  1,  1, childCells));
}
```

D. *Road Intersection*

Every node checks the roadCutters array before it is divided. IntersectsAnyRoad() stops as soon as one collider overlaps the current node. RoadIntersectsBounds() projects the axis-aligned block and the possibly rotated road rectangle onto four axes on the XZ plane. The road intersects the block when the projected intervals overlap on every axis. The main idea of the road intersection and its meaning to the subdivision can be viewed by the diagram below.

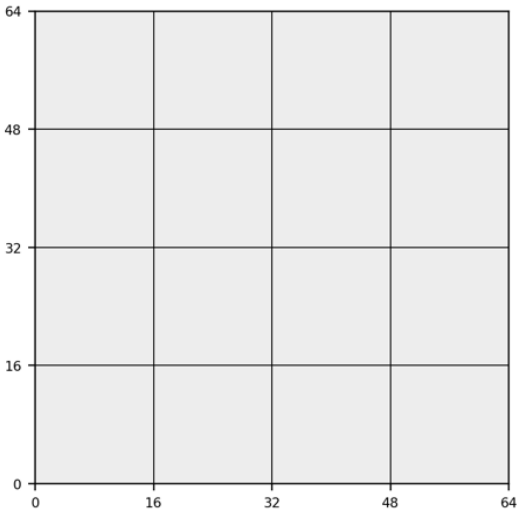

Fig 2. No Road Subdivision Illustration

Above shows a case where a default town has no road. None of the root blocks intersect a road collider. Every block therefore stops at the first level and remains at its original size.

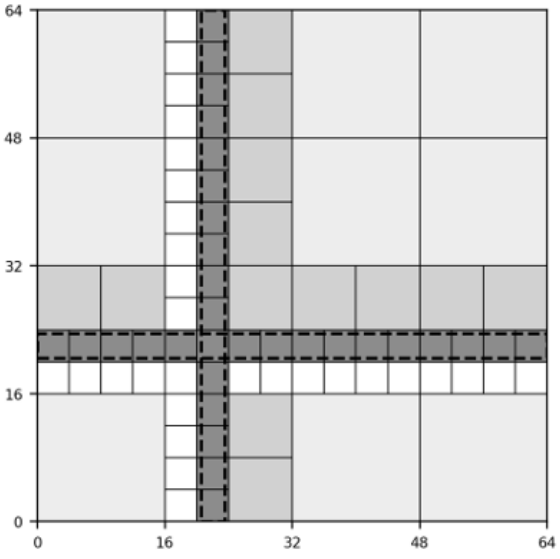

Fig 3. Cross Road Subdivision Illustration

Meanwhile, in the figure above, shows 2 road intersecting horizontally and vertically. Subdivision on root blocks that intersects with the road collider happens. The blocks around the road intersection receive the deepest subdivision because they overlap two road colliders. Blocks farther from the roads remain large.

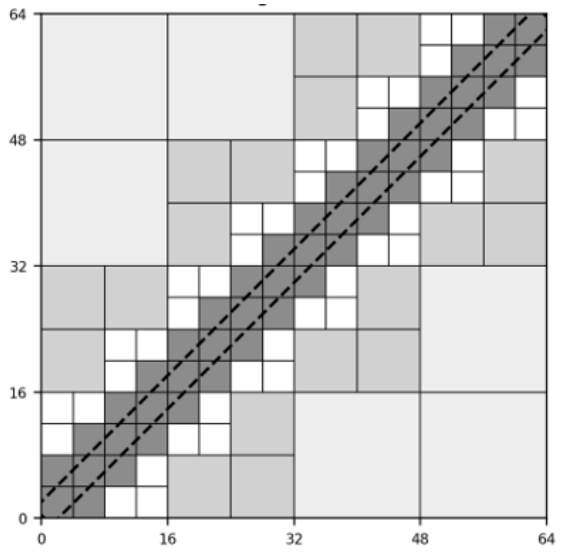

Fig 4. Diagonal Road Subdivision Illustration

The same sense of occurrence happen on diagonal road placements, which create more subdivisions since there is more root blocks that intersects with the road. The quadtree still divides every block using axis aligned square quadrants. Which cause the road boundary is represented by a jagged grid arrangement of smaller leaves.

E. Asset Selection

After recursion, the leaves are processed one by one. Road leaves receive a road surface. Clean leaves receive a ground object and a building prefab selected from their side length in cells.

```
if (leaf.isRoad)
{
    SpawnClippedRoadSurfaces(leaf.bounds);
    lastRoadLeafCount++;
    return;
}

if (leaf.cells >= 4)
    SpawnFitted(ChoosePrefab(
        palette.bigBuildings), leaf.bounds,
        "Big Building");
else if (leaf.cells >= 2)
    SpawnFitted(ChoosePrefab(
        palette.mediumBuildings), leaf.bounds,
        "Medium Building");
else
    SpawnFitted(ChoosePrefab(
        palette.smallHouses), leaf.bounds,
        "Small Building");
```

The asset catalogue itself is stored as a interchangeable scriptable object

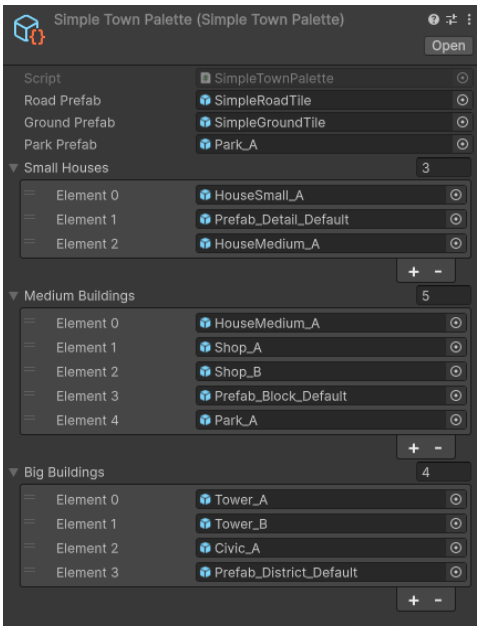

Fig 5. Asset Storing Through A Scriptable Object

Figure above shows the current setup of asset used in this project. SpawnFitted() reads the prefab renderer bounds and applies one uniform scale so the object fits inside the leaf. The optional smooth road mode clips the road rectangle to the leaf before creating the road mesh. The Inspector also records the number of leaves, road leaves, building tiers, and generation time after each run.

IV. Result and Testing

Testing was conducted directly in the Unity Editor using three road configurations while maintaining the same town dimensions, subdivision parameters, random seed, prefab palette, and park probability. The recorded generation time includes scene cleanup, recursive subdivision, road intersection testing, prefab instantiation, asset fitting, and road mesh generation. Each reported value represents one observed generation run.

Parameter	**Value**
Town size	16 x 16 cells
Cell size	4 Unity units
Starting block	4 x 4 cells
Minimum block	1 x 1 cell

Table 1. Testing Configuration

Config	Leaves	Road Leaves	Large	Medium	Small
No Road	16	0	12	0	0
Cross And Diagonal	166	92	3	10	46
Multi Angle Diagonal	136	69	3	15	37

Table 2. Generation Result For Each Configuration

A. *No Road Configuration*

The baseline has no road. All sixteen 4 x 4 root blocks stay as large leaves. Creating exactly 16 leaves and 16 big buildings asset chosen from the catalog.

Fig 6. No-Road Town Configuration Result

The top-down screenshot shows a regular (4x4) arrangement of large lots. The perspective screenshot shows that each asset used occupies the same grid size. Differences in building model and rotation are produced by seeded prefab selection, while the spatial size remains uniform throughout the town. Compared with a uniform (1x1) cell representation containing 256 leaves, the adaptive structure uses only sixteen leaves. The observed generation time is 1.96 ms.

B. *Cross And Diagonal Road*

Fig 7. Cross and Diagonal Road Configuration Result

The second configuration uses crossing roads together with a diagonal road segment. Root blocks intersected by these road cutters are recursively subdivided, while blocks outside the road influence remain at their original size.

The configuration uses fewer leaves than the 256 leaf baseline. The generated tree contains 50 internal subdivision nodes and 216 processed nodes in total. The observed generation time is 23.04 ms, which is the highest among the three configurations.

The top-down screenshot shows the adaptive property of the quadtree. Large building lots remain visible near the outer sections of the town. Areas closer to the crossroad contain medium and small lots because their parent nodes intersect the road and are divided to a greater depth.

A limitation visible in this configuration is that building orientation remains independent of road direction. Buildings receive random rotations in multiples of 90 so their entrances are not guaranteed to face the nearest road. Though, that is to be expected, since the scope of this project is to show Divide and Conquer and not an advanced town generation.

C. *Multi-Angle Diagonal Road*

The third configuration uses several diagonal road cutters with different orientations. Compared with the cross-road configuration, these roads pass through a larger number of starting blocks and intersect more child regions

Fig 8. Multi-Angle Diagonal Road Configuration Result

The top-down screenshot shows concentrated refinement along each diagonal corridor. Regions that remain outside all road colliders preserve their (4x4) cell size and can contain large buildings. Regions intersected only at the first subdivision level may terminate as (2x2) cell leaves and

receive medium buildings. Regions directly crossed by roads continue to the (1x1) cell level. The observed generation time is 18.88 ms. Although this configuration contains several diagonal roads, it creates fewer leaves than the cross-and-diagonal configuration. Subdivision quantity therefore depends on the total spatial overlap between the roads and the quadtree regions

Fig 10. Example Result On Large Grid Configuration

Above is the full potential of this project. Truly, the aim is to create a city composition from subdividing boundaries. A limitation of the project should be mentioned, as this project only handles a flat XZ layout and does not subdivide height. Road positions are supplied manually through BoxCollider objects. The generator does not assign zoning, simulate traffic, reserve sidewalks, or evaluate relationships between neighboring buildings. Building orientation uses random quarter turns instead of the nearest road direction. Visual repetition can also occur when a prefab category contains only a few models.

V. Conclusion

Divide and Conquer provides a clear structure for road-driven town-space partitioning. The town region is divided into four smaller regions, each child is conquered through the same overlap test, and the resulting leaves are combined into a procedural composition. A quadtree-inspired hierarchy allows road-adjacent space to reach fine resolution while unaffected space remains coarse.

The Unity implementation connects BoxCollider road cutters, recursive Bounds subdivision, leaf classification, and prefab instantiation in one traceable pipeline. Qualitative tests show that straight roads, boundary roads, and intersections create different local subdivision patterns and corresponding asset scales. The method is suitable for a small adaptive town prototype and provides a foundation for footprint-aware buildings, road-facing placement, lot merging, and larger spatial indexes.

Appendix

Github Repository: Arbane557/Stima-DnCTown

Acknowledgment

The author extends sincere appreciation to Ir. Rila Mandala, M.Eng., Ph.D. and Dr. Ir. Rinaldi Munir, M.T., whose invaluable guidance, support, and encouragement throughout the preparation of this manuscript have been truly instrumental. His expertise, and unwavering dedication to teaching have greatly enriched the author's understanding.

References

[1] R. Munir, "Algoritma Divide and Conquer, Bagian 1," IF2211 Strategi Algoritma, Program Studi Teknik Informatika, Institut Teknologi Bandung, 2026. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2025-2026/07-Algoritma-Divide-and-Conquer-(2026)-Bagian1.pdf. [Accessed: Jun. 19, 2026].

[2] OpenDSA Project Contributors, "15.4. The PR Quadtree," CS3 Data Structures & Algorithms. [Online]. Available: https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/PRquadtree.html. [Accessed: Jun. 19, 2026].

[3] Kenney, "Modular Buildings." [Online]. Available: https://www.kenney.nl/assets/modular-buildings. [Accessed: Jun. 19, 2026].

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 19 Juni 2026

Raysha Erviandika Putra 13524050